Memory-Efficient Optimization of Gyrokinetic Particle-to-Grid Interpolation for Multicore Processors

Kamesh Madduri[†], Samuel Williams[†], Stéphane Ethier[‡], Leonid Oliker[†]
John Shalf[†], Erich Strohmaier[†], Katherine Yelick[†⋆]
CRD/NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
Princeton Plasma Physics Laboratory, Princeton, NJ 08543
EECS Department, University of California at Berkeley, Berkeley, CA 94720

ABSTRACT

We present multicore parallelization strategies for the particle-to-grid interpolation step in the Gyrokinetic Toroidal Code (GTC), a 3D particle-in-cell (PIC) application to study turbulent transport in magnetic-confinement fusion devices. Particle-grid interpolation is a known performance bottleneck in several PIC applications. In GTC, this step involves particles depositing charges to a 3D toroidal mesh, and multiple particles may contribute to the charge at a grid point. We design new parallel algorithms for the GTC charge deposition kernel, and analyze their performance on three leading multicore platforms. We implement thirteen different variants for this kernel and identify the best-performing ones given typical PIC parameters such as the grid size, number of particles per cell, and the GTC-specific particle Larmor radius variation. We find that our best strategies can be $2\times$ faster than the reference optimized MPI implementation, and can substantially reduce the MPI memory footprint. Our analysis provides insight into desirable architectural features for high-performance PIC simulation codes.

1. INTRODUCTION

Full long-range particle-particle force interaction simulations carry an unacceptably high computational complexity — $O(N^2)$ in the number of particles. To that end, a number of techniques have been developed to mitigate the computation time. The Particle-in-Cell (PIC) method is a widely-used technique and can be applied to the simulation of plasmas. This method solves the kinetic equation by following the individual trajectories of the plasma's constituent charged particles. Since electric and magnetic forces are long-range forces, the direct calculation of the binary interaction between each pair of particles quickly becomes prohibitive for large number of desired particles. Furthermore, the direct interaction automatically includes the full collision dynamics between the particles, which overwhelms the underlying collective physics unless an extremely large number of particles is used to simulate the plasma. In order to avoid these pitfalls, the PIC method uses an auxiliary grid to approximate the particles' charge density as it varies in space and time. Using this distribution, it solves Poisson's equation to calculate the electromagnetic potential at any point in space. The particles are then accelerated by the potential. This calculation is only $O(N)$ and has the advantage of removing the effects of close encounter collisions, allowing the simulation of the collective effects in the plasmas with only a small number of particles compared to the number in real plasmas.

Although the PIC approach drastically reduces the simulation computational requirements, achieving good parallel- and architectural-efficiency is far more challenging than the full $O(N^2)$ calculation. The particle-to-grid interpolation step of charge deposition in particular is a major performance bottleneck [11]. Unlike simple histograms, particles update the bounding grid points each with a fraction of their charge. Randomly localized particles may make poor use of caches. The challenges of this deposition step become far more significant when one contemplates the RISC nature of modern processors. Floating-point increment is typically not an atomic operation. This can prohibit or impede unrolling or parallelizing the particle array as multiple particles may attempt to increment the same grid points.

The Gyrokinetic Toroidal Code (GTC) [7, 4] was developed to study the global influence of microturbulence on particle and energy confinement. It is a three-dimensional, fully self-consistent PIC code which solves the kinetic equation in a *toroidal* geometry. The charge update scheme in GTC differs from traditional PIC codes. In the classic PIC method, a particle is followed directly and the charge is distributed to its nearest neighboring grid points. However, in the gyrokinetic PIC approach used in GTC, the fast circular motion of the charged particle around the magnetic field lines is averaged out and replaced by a charged ring. For the wavelengths of interest for low frequency microturbulence studied by GTC, it uses four points on the charged ring [6] to describe the non-local influence of the particle orbit. In this way, the full influence of the fast, circular trajectory is preserved without having to resolve it. However, this scheme inhibits straightforward shared-memory parallelism even further, since the update positions need to be computed for each particle and at each time step. Further, multiple particles may concurrently attempt to deposit their change onto the same grid point.

1.1 Our Contributions

Our work focuses on multicore parallelization strategies for the charge deposition kernel of GTC. The memory requirements in the current MPI implementation of the charge deposition kernel scale proportionally with the number of cores/threads in a single node, as the grid is replicated on all processes in the corresponding toroidal domain to allow for concurrent updates. However, this becomes a scalability concern on multi- and many-core systems simulating large or dense toroidal grid instances. We present several grid decomposition and synchronization schemes to partition the charge deposition work among multiple threads of execution while limiting the memory requirements of our approaches to typically within triple the serial execution's footprint. On the three multicore machines on which we analyze performance, we observe that the best Pthreads implementation consistently outperforms the reference MPI code.

The decomposition schemes and optimizations discussed in this paper are applicable to other PIC codes as well. GTC is arguably one of the more challenging PIC codes to optimize, due to the underlying toroidal geometry and the gyrokinetic averaging scheme. We give a brief overview of GTC in the next section, before exploring the charge deposition kernel in detail.

2. GYROKINETIC TOROIDAL SIMULATION

As the global energy economy transitions from fossil fuels to cleaner alternatives, nuclear fusion becomes an attractive potential solution for satisfying growing needs. Fusion, the power source of the stars, has been the focus of active research since the early 1950s. While progress has been impressive — especially for magnetic confinement devices called tokamaks — the design of a practical power plant remains an outstanding challenge. A key topic of current interest is microturbulence, which is believed to be responsible for the experimentally observed leakage of energy and particles out of the hot plasma core. Understanding and controlling this process is of utmost importance for operating current devices and designing future ones. This goal led to the design of GTC to study the global influence of microturbulence on particle and energy confinement.

GTC solves the non-linear gyrophase-averaged Vlasov-Poisson equations [6] for a system of charged particles in a self-consistent, self-generated electrostatic field. The geometry of the system is that of a torus with an externally imposed equilibrium magnetic field, characteristic of toroidal fusion devices. By using the PIC method, the non-linear partial differential equation describing the motion of the particles in the system becomes a simple set of ordinary differential equations that can be easily solved in the Lagrangian coordinates. The self-consistent electrostatic field driving this motion is calculated via the PIC approach, by using a grid where each particle deposits its charge to a limited number of neighboring points according to its range of influence.

The current production version of GTC scales well with the number of particles on some of the largest supercomputing systems [10, 9, 5]. It achieves this by using multiple levels of parallelism: a 1D domain decomposition in the toroidal dimension (long way around the torus geometry), a multi-process particle distribution within each one of these toroidal domains, and a loop-level multitasking implemented with OpenMP directives [1]. The local grid within a toroidal domain is replicated on each MPI process within that domain and the particles are randomly distributed to cover that whole domain. The grid work, which comprises of the field solve and field smoothing, is performed redundantly on each of these MPI processes in the domain. Only the particle-related work is fully divided between the processes. This is not an issue as long as the grid work remains small compared to the particle work, which is the case for most of the GTC simulations carried out to date.

3. EXPERIMENTAL SETUP

In this section, we describe in detail the machines used in this study, and our benchmarking methodology. We select three leading multicore designs to explore the benefits of our threaded implementations of the charge deposition kernel across a variety of architectural paradigms. To mitigate the impact of limited bandwidth, long latency, and coherency networks, we limit ourselves to dual-socket SMPs. A summary of their architectural parameters is provided in Table 1.

Intel Nehalem: The recently released Nehalem is the latest enhancement to the Intel "Core" architecture, and represents a dramatic departure from Intel's previous multiprocessor designs. It abandons the front-side bus (FSB) in favor of on-chip memory controllers. The resultant QuickPath Interconnect (QPI) inter-chip network is similar to AMD's HyperTransport (HT), and it provides access to remote memory controllers and I/O devices, while also maintaining cache coherency. Nehalem is novel in two other aspects: support for two-way simultaneous multithreading (SMT) and TurboMode. The latter allows one core to operate faster than the nominal clock rate under certain workloads. On our machine, TurboMode is disabled due to its inconsistent timing behavior.

The system used in this study is a dual-socket, quad-core 2.66 GHz Xeon X5550 (Gainestown) with a total of 16 hardware thread contexts. Each core has a private 32 KB L1 and a 256 KB L2 cache, and each socket instantiates a shared 8 MB L3 cache. Additionally, each socket integrates three DDR3 memory controllers operating at 1066 MHz, providing up to 25.6 GB/s of DRAM bandwidth to each socket. In comparison to the Barcelona system used in this paper, Nehalem has a similar floating-point peak rate but a significantly higher memory bandwidth and cache.

AMD Opteron 2356 (Barcelona): The Opteron 2356 (Barcelona) is AMD's quad-core processor offering. Each Opteron core runs at 2.3 GHz, has a 64 KB L1 cache, and a 512 KB L2 victim cache. In addition, each chip instantiates a 2MB L3 quasi-victim cache that is shared among all four cores. Each Opteron socket includes two DDR2-667 memory controllers providing up to 10.66 GB/s of raw DRAM bandwidth. Sockets are connected via a cache-coherent HT link creating a coherency and NUMA network for this 2 socket (8 core) machine.

Sun UltraSparc T2+ (Victoria Falls): The Sun "UltraSparc T2 Plus", a dual-socket $\times$ 8-core SMP referred to as Victoria Falls, presents an interesting departure from mainstream multicore processor design. Rather than depending on four-way superscalar execution, each of the 16 strictly in-order cores supports two groups of four hardware thread contexts (referred to as Chip MultiThreading or CMT) — providing a total of 64 simultaneous hardware threads per socket. Each core may issue up to one instruction per thread group assuming there is no resource conflict. The CMT approach is designed to tolerate instruction, cache, and DRAM latency through fine-grained multithreading. Victoria Falls

Table 1: **Architectural details of parallel platforms.**

Core Architecture	AMD Barcelona	Intel Nehalem	Sun Niagara2
Type	superscalar out of order	superscalar out of order	HW multithreaded dual issue
Clock (GHz)	2.30	2.66	1.16
Double-precision GFlop/s	9.2	10.7	1.16
L1 Data Cache	64 KB	32 KB	8 KB
private L2 cache	512 KB	256 KB	—
System	**Opteron 2356 (Barcelona)**	**Xeon X5550 (Gainestown)**	**UltraSparc T5140 (Victoria Falls)**
# Sockets	2	2	2
Cores(Threads) per Socket	4(4)	4(8)	8(64)
Primart memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading
Shared last-level cache	2×2 MB (shared by 4 cores)	2×8 MB (shared by 4 cores)	2×4 MB (shared by 8 cores)
DRAM Capacity	16 GB	12 GB	32 GB
DRAM Pin Bandwidth (GB/s)	21.33	51.2	42.66(read) 21.33(write)
Double-precision GFlop/s	73.6	85.3	18.7

has no hardware prefetching, and software prefetching only places data in the L2 cache. Multithreading may hide instruction and cache latency, but may not fully hide DRAM latency. Our machine is clocked at 1.16 GHz, does not implement SIMD, but has an aggregate 64 GB/s of DRAM bandwidth in the usual 2:1 read:write ratio associated with FBDIMM. As such, it has significantly more memory bandwidth than either Barcelona or Nehalem, but has less than a quarter the peak flop rate. With 128 hardware thread contexts, this Victoria Falls system poses parallelization challenges that we do not encounter in the Gainestown and Barcelona systems.

3.1 Methodology

To analyze multicore performance of the GTC particle-to-grid interpolation step, we extract the key computation in an optimized GTC MPI implementation to create a stand-alone PIC charge deposition benchmark. The data representation and the computation in the initial version of this benchmark are identical to the reference MPI code.

Problem Instances: There are several input parameters in GTC to describe the test simulation. The ones most relevant to the charge deposition kernel are the size of the discretized toroidal grid, the total number of particles, and the Larmor radius distribution of the particles for four-point gyrokinetic averaging. Often one replaces the number of particles with the average particle density as measured in the ratio of particles to grid points (labeled as *micell*). Three coordinates describe the position of a particle within the discretized torus: ζ (*zeta*, the position in the toroidal direction), ψ (*psi*, the radial position within a poloidal plane), and θ (*theta*, the position in the poloidal direction within a toroidal slice). The corresponding grid dimensions are *mzeta*, *mpsi*, and *mthetamax*. In this paper, we explore four different grid problem sizes, labeled *A, B, C, D*, and vary the particle density from 5 and 100. Table 2 lists these settings, and these are similar to ones used in prior experimental studies and GTC production runs [1, 5].

The Larmor radius of a particle ρ is dependent on the value of the particle's magnetic moment and the local value of the magnetic field B. The magnetic moment of the particle, which does not vary during the simulation, is given by $B \cdot \rho^2$. As B changes with the position of the particle, ρ also varies. For all four GTC problem sizes we use in this

Table 2: **The GTC test problem settings used in our experimental study.**

Problem Size	A	B	C	D
mzeta	1	1	1	1
mpsi	90	192	384	768
mthetamax	640	1408	2816	5632
mgrid	32449	151161	602695	2406883
Total Particles (micell=5)	0.16M	0.76M	3M	12M
Total Particles (micell=100)	3M	15M	60M	241M

study, the maximum Larmor radius (a function of several other GTC parameters) turns out to be roughly $mpsi/16$. Further, the radii values are initially chosen from a uniform random distribution. In Section 4 and 5, we discuss how this choice affects the parallelization strategies we employ.

Distribution and Movement of Particles: As we are only studying charge deposition in this paper, as opposed to the entire GTC application, we assume that particles do not move along the toroidal direction. As such a given particle will always update the same locations. However, we do not explicitly exploit this characteristic. To exercise different architectural features, we experiment with two initial particle distributions: ψ-sorted and random. Unless otherwise stated, any data presented is based on the ψ-sorted distribution. Given the possible range of variation along the θ direction, it is likely that two logically adjacent particles in a ψ-sorted distributed can be quite far apart in the toroidal grid. Whether running the MPI or threaded version, we assume that a set of MPI processes/threads own exactly one plane in *zeta* and maintain one ghost copy of the neighboring plane. This is consistent with typical GTC parameters settings in production runs.

Reference MPI version: The MPI implementation statically partitions the particles within a toroidal segment among the participating MPI tasks. Each task maintains a private copy of the two bounding poloidal planes, and may thus deposit charge independently. An MPI reduction is performed to reduce the N copies of the grid to one. We explored a range of degrees of parallelism when running the MPI implementation on our SMPs. When benchmarking, we run successive iterations of local charge depositions and MPI reductions.

PThreads versions: To evaluate performance of the threaded implementations, we employ a SPMD-inspired threading model in which communication of particles or updates to grid values is handled through shared memory rather than message passing. Typically, we create N threads, statically partition the particles, allow them to initialize their data, and then benchmark repeated charge depositions.

Timing and GFlop/s: We use high-precision cycle counters on the different machines to measure the total time for ten iterations of charge deposition. We manually counted the number of floating point operations (flops) within the function. We express average GFlops/s as the ratio of total flops to the average execution time.

Build software environment: We build our Pthreads codes on the three machines with the GNU C compiler and aggressive architecture-specific optimization flags. We use GNU Fortran90 compiler with identical optimization flags to compile the reference GTC Fortran code, and the MPICH2 library (version 1.0.8) with a fast shared memory communication device (`ch3:shm` on the Nehalem and Victoria Falls systems, and `ch3:nemesis` on the Barcelona system).

4. GTC CHARGE DEPOSITION KERNEL

Charge deposition is easily GTC's most challenging kernel to optimize. The overall memory access pattern of this kernel may vary across time steps, and is dependent on several GTC simulation parameters. Elements of the charge density grid are updated by each particle slightly differently at each time step, as the updates are based on the particle's position, velocity, and magnetic moment values. Further, each particle can update up to 32 possibly non-contiguous locations when applying the four-point gyro-averaging scheme, leading to significant performance variation across combinations of cache-based superscalar architecture, grid size, and the number of particles per cell. In this section, we discuss shared-memory parallelization strategies and optimizations that are applicable for a range of grid size and particles-per-cell configurations.

4.1 Overview

Each iteration of the charge deposition kernel performs roughly 180 floating point operations, and the computation proceeds in two phases. First, given a particle's current spatial coordinates, we identify the two bounding poloidal planes to interpolate charge values to. Based on the particle's Larmor radius, we determine up to sixteen grid positions to update on each plane, and the corresponding charge density increments. Note that if the particle's Larmor radius is smaller than the radial grid spacing, several grid update positions may overlap. This *address calculation* phase requires 115 flops, including a square-root operation. Next, we access the density array to increment charge densities at 32 positions, and perform two flops per update. We refer to this as the *density update* phase.

The toroidal grid and the gyrokinetic averaging scheme are the two key computational aspects of GTC that distinguish it from other PIC applications. Both these characteristics introduce variability in charge density update positions for a particle, and the GTC particle-grid interpolation step typically demonstrates a lower cache locality compared to PIC codes that use a rectilinear grid discretization [3]. Further, the particle Larmor radii distribution impacts load balancing when parallelizing the density update phase.

4.2 Parallelization

In order to parallelize the charge deposition computation, we employ a straightforward partitioning of the particle array. Thus a given particle is owned and exclusively operated on by one and only one thread. This lets us exploit spatial locality in particle array accesses for the address calculation phase, and also results in a balanced partitioning of computation among threads. However, there is no guarantee that the grid points bounding a given particle will be updated exclusively by the thread processing the particle. Thus, a synchronization lock must be employed to provide mutual exclusion when there are multiple threads accessing these grid points. Locks are known to be slow. As such, there are several strategies to mitigate this performance impact. These can be categorized into the orthogonal concepts of grid decomposition and synchronization mechanisms.

4.3 Grid Decomposition

In Figure 1, we illustrate four possible grid decomposition schemes that are applicable to this kernel. Figure 1(a) is the purest shared memory implementation where there is only one copy of the grid, and all threads must contend for exclusive access. At the other extreme is the approach taken in the MPI implementation. As shown in Figure 1(d), each thread or process maintains a local duplicate of the grid and no synchronization is required for updates. The memory requirements scale with the number of cores for this approach, a seemingly inappropriate solution in the multicore era.

It is evident that any static grid partitioning will benefit this kernel only when the particles are ordered and processed based on their grid position. The ordering can range from a full sort (sorting by r, *theta*, and *zeta* coordinates), to simple binning [2] in the radial direction. We demonstrate in Section 5 that parallel performance is severely affected if particle positions are fully randomized. Occasional radial particle binning is essential to ensure consistently high performance for charge deposition, as well as other GTC kernels [8]. Thus, we assume a ψ-sorted particle distribution in this paper, and will explore performing radial binning in conjunction with charge deposition in future work. It is also reasonable to assume that binning is unnecessary for every simulation time step, as the particle position variation in the radial direction is considerably less than the θ and ζ directions.

On multicore architectures, we strive for a middle ground in which we create one or more auxiliary copies of the grid to accelerate charge deposition. Our second approach, shown in Figure 1(b), adds one auxiliary grid that is partitioned into disjoint annuli exclusively owned and updated by a unique thread. We label this approach the *Partitioned Grid*. For each of the 32 grid points to be updated, if the update location lies within the thread's exclusively owned grid, the increment may be performed without any synchronization mechanism. However, if the grid point lies outside the thread's exclusively owned grid, the thread will write to the shared grid using a synchronization mechanism to obtain exclusive access. Clearly, after all grid updates have been performed, we have a thread barrier, and the two (shared and auxiliary) copies of the grid must be reduced to one.

In the partitioned grid approach, we decompose the grid into annuli such that each region encompasses approximately the same number of grid points. However, note that the number of grid points per annulus varies, and annuli typ-

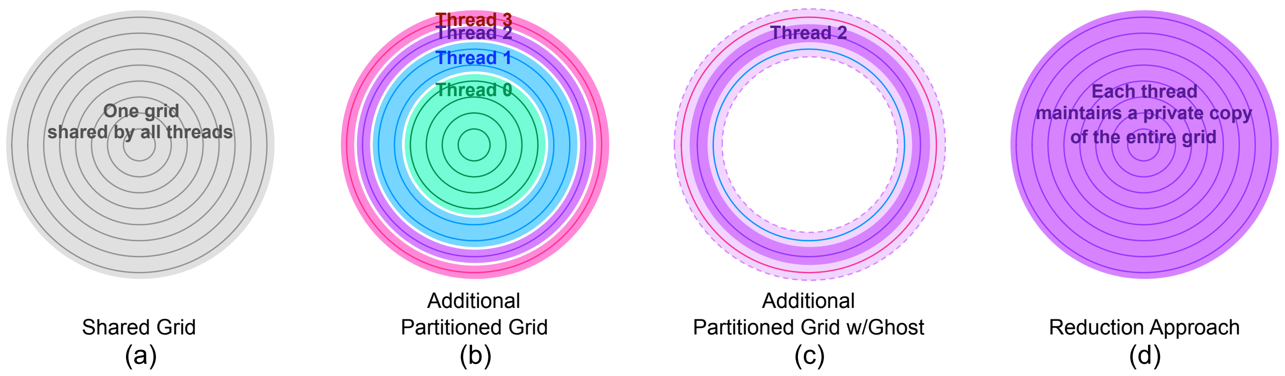

One grid shared by all threads
Shared Grid
(a)
Thread 3
Thread 2
Thread 1
Thread 0
Additional Partitioned Grid
(b)
Thread 2
Additional Partitioned Grid w/Ghost
(c)
Each thread maintains a private copy of the entire grid
Reduction Approach
(d)

Figure 1: An illustration of the four different grid decomposition/replication strategies we utilize in the Pthreads implementations. Note, in (c) thread 2 maintains a copy of parts of thread 1 and 3's domains.

ically have more points as we move radially outward. For parallel runs at high concurrencies, where the number of threads is roughly equal to the number of radial surfaces (*mpsi*), we may further partition the grid points in a single annulus among multiple threads for load balancing. We label this optimization as *Fine partitioning* and expect it to be helpful in case of smaller grid instances.

As GTC performs a four-point gyrokinetic averaging, it is usually the case that the updated grid points span several radii. Thus it is quite possible that threads will often access the shared grid even when the particles have been partially sorted. To that end, as shown in Figure 1(c), we add an auxiliary grid in which the subgrids are overlapping. Each subgrid is radially extended inward and outward with a ghost surface. Such an approach helps to mitigate synchronization overhead, but requires slightly more memory, and a slightly more complex reduction. We label this approach the *Partitioned Grid with Ghost Surfaces*.

The efficacy of the grid decomposition schemes is dependent on the maximum Larmor radius value, as well as the distribution of Larmor radii values. As we note in the previous section, for the problem sizes and GTC simulation parameters we use, the maximum Larmor radius is roughly $mpsi/16$ and is *independent* of the radial position of the particle. This setting gives us an upper bound on the reduction in shared grid accesses we can achieve using any of the above grid decomposition schemes.

4.4 Synchronization Mechanisms

We next discuss possible synchronization mechanisms to access the shared grid in the above decomposition schemes. As shown in Figure 2, we explore four different methods for providing exclusive access. Although one might naïvely couple any of these methods with any grid decomposition strategy, there are only 13 legitimate combinations — Figure 1(a–c) × Figure 2(a–d), and the *fully replicate and reduce algorithm* of Figure 1(d).

Figure 2(b) illustrates a coarse-grained locking mechanism implemented using Pthreads mutual exclusion routines in which we lock only on one of the concentric $theta \times zeta$ cylindrical surfaces — essentially a series of rings extruded in *zeta*. Although each thread must acquire only 8 successive locks per particle update, concurrency is limited to the number of concentric cylinders (*mspi*). This coarse-grained locking scheme is reasonable under the assumption that particles are binned according to their radial coordinate.

As shown in Figure 2(c), we may reduce the granularity of locking to all values in *zeta* for particular coordinate in *theta* and *psi*. Such an approach dramatically increases concurrency, but mandates that each thread acquire up to 16 locks per particle.

The final locking approach is fine-grained locking and locks only one grid point. This approach doubles the potential concurrency, but also doubles the number of locks per particle.

One should observe that we strive for atomicity in updates, and not a particular ordering. To that end, in addition to the lock-based approaches, we hand-code x86 assembly and utilize SPARC intrinsics to implement an atomic floating-point increment. Such a method should have lower overhead than a Pthreads-based mutex lock and provide atomicity, but will still be slower than the RISC load–increment–store operation. When using this approach, each thread performs 32 atomic floating-point increments per particle (illustrated in Figure 2(a)).

It should be reiterated that when updating an exclusively owned copy of a grid point, neither locks nor atomic increments are required. Additionally, one may express 8-way memory-level parallelism per thread (the 8 grid points boxing a particle).

As one migrates from Figure 2(b) to Figure 2(d), the overhead per charge deposition increases due to increased number of locks per particle. However, this can be mitigated with increased concurrency. We generally expect the hand-coded atomic increments to provide the highest concurrency, but probably not the lowest overhead per particle. As different architectures have different concurrency demands and different overheads for locking and atomic implementations, we explore all combinations for each processor.

4.5 Serial Optimizations

Several serial code optimizations are applicable to the Pthreads charge deposition code variants. The particle position, velocity, and other auxiliary data (in total six double-precision values per particle) are stored in an array-of-structures representation in the original Fortran code. Since we require only five of the six values in the charge deposition kernel, we switch to a structure-of-arrays representation in the C benchmark. We flatten all Fortran multi-dimensional auxiliary grid arrays in the C code and align them to cache line boundaries during initialization. Next we fuse the loops corresponding to the address calculation and density update

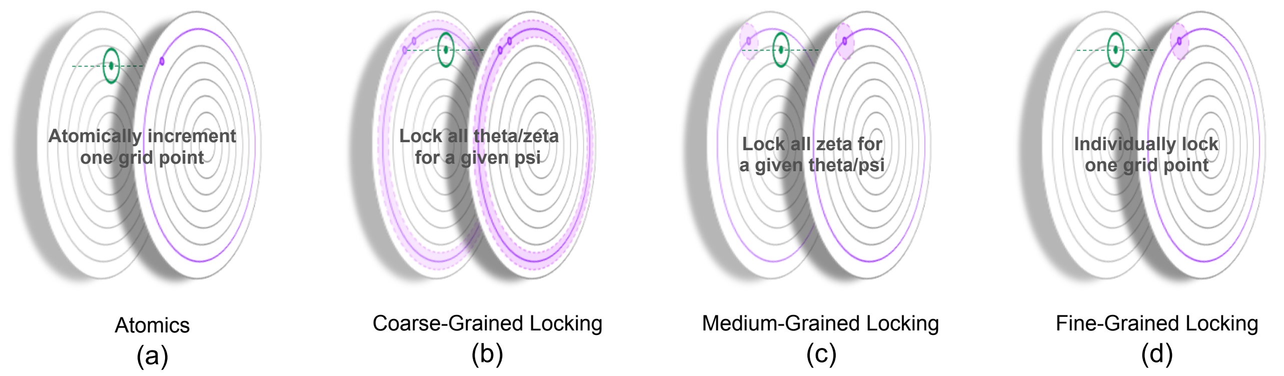

Atomically increment one grid point
Lock all theta/zeta for a given psi
Lock all zeta for a given theta/psi
Individually lock one grid point
Atomics
(a)
Coarse-Grained Locking
(b)
Medium-Grained Locking
(c)
Fine-Grained Locking
(d)

Figure 2: An illustration of possible synchronization strategies for the Pthreads implementations. Purple shaded regions denote the lock granularity.

steps to maximize particle data reuse. We use SSE2 instructions on the Barcelona and Nehalem systems to increment charge densities of grid points that are consecutively laid out in ζ. The charge density step involves 20 stores per particle to five different arrays, and these values are used in latter simulation routines. We observe up to a 30% increase in performance if we *do not* time these stores (not shown in figures). This gives us an estimate of realizable performance improvement with data reorganization. We also find that the single square-root instruction in the address calculation phase does not significantly impact performance. The kernel is about 2-3% faster if we precompute this value.

5. EXPERIMENTAL RESULTS

In this section, we perform an extensive analysis of the performance and memory utilization characteristics of the various threaded optimizations on the charge deposition kernel, for the class B problem size with a density of 5 particles per grid point. Unless otherwise noted, the particles have been initially sorted to provide locality to partitioned grid strategies. We follow this analysis with a performance survey across a wide range of the problem configurations detailed in the Section 3 for memory-efficient and random particle distribution restrictions.

5.1 Decomposition, Synchronization, and Optimization Performance

Figure 3 shows performance as a function of the 13 possible combinations of grid decomposition and synchronization schemes stacked with serial optimizations for the class B problem size. *micell* is set to 5 in all cases. All data points show the best performance with any concurrency. The dashed line represents the performance of the reference MPI implementation for the same problem configuration.

Although Nehalem has the same number of cores, and only a moderately higher flop rate, we see the substantially larger cache and higher bandwidth yield twice the performance of Barcelona. While Victoria Falls has as much cache as Barcelona, and substantially better bandwidth, its much lower flop rate results in it being substantially slower than either Barcelona or Nehalem.

We observe that the simple replicate and reduce strategy provides the best performance on all three systems for this problem instance. The partitioned grid with ghost surfaces decomposition schemes perform substantially better than the pure shared grid decompositions. The performance improvement is directly correlated with the reduction in shared grid increments. On the 128-way threaded Victoria Falls system, the $mpsi/128$-annuli thick ghost zones do not give us a substantial benefit, as the maximum Larmor radius is roughly $mpsi/16$.

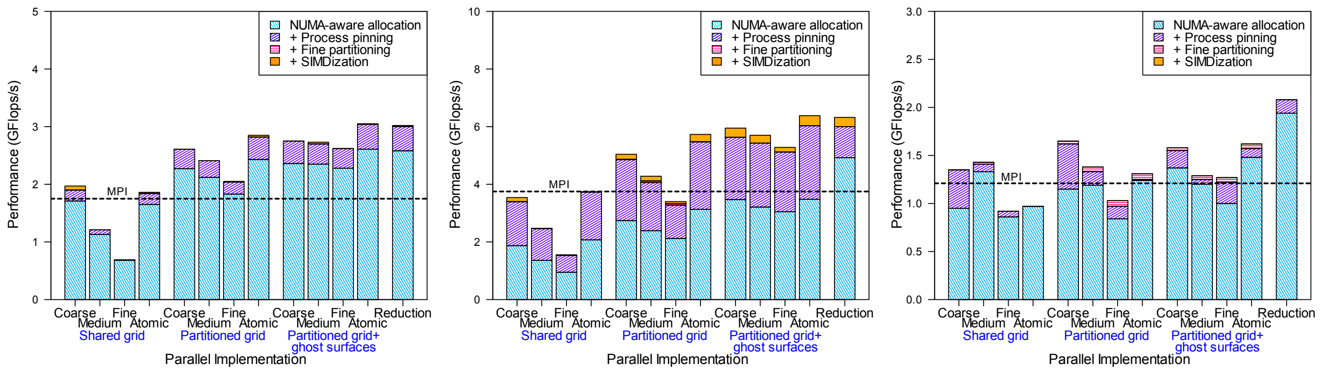
Given the relatively low thread-level parallelism employed on the x86 systems, it should come as no surprise that coarse-grained locking provided superior performance when compared to either medium- or fine-grained locking. Essentially, the dramatically reduced locking overhead per particle wins out over reduced concurrency. Also, since the particles are ψ-sorted, the coarse-grained lock contention is low on the x86 systems. A surprising result occurs when using atomic increments on the x86 processors. We see that performing up to 32 atomic increments per particle provides better performance than 8 coarse-grained locks. We use an atomic compare-and-swap instruction to implement the floating point increment, and the same strategy can be applied to implement a mutex spinlock which should perform well under low contention. This observation suggests that x86 Pthreads mutex library routines are not the best possible implementations. Interestingly, the SPARC mutex routines deliver performance much closer to the intrinsic-based atomic increments.

We also observe that for this kernel, process pinning is perhaps the most valuable serial optimization. *Fine partitioning* gives a moderate performance improvement on the Victoria Falls system. SIMDization provides some benefit on Nehalem, which may be due to the lower double-precision Flop/byte ratio of this system when compared to Barcelona.

Typically, the single shared grid Pthreads implementation achieved performance on parity with the MPI implementation. However, the decompositions that performed some replication substantially exceeded MPI performance. For instance, the memory-efficient partitioned grid with ghost flux surfaces and atomic increments scheme performs 1.5×, 1.7×, and 1.2× faster that the MPI code on the Nehalem, Barcelona, and Victoria Falls systems respectively.

5.2 Scalability within an SMP

Figure 4 shows the scalability of the underlying architecture for the best performing synchronization method on the grid decompositions presented in Figure 3. For the Pthreads implementations, we enumerate hardware thread contexts so that when ramping up the number of threads, we exploit multithreading within a core, then multicore on chip,



(a) Barcelona performance (b) Nehalem performance (c) Victoria Falls performance

Figure 3: Performance as a function of grid decomposition, synchronization, and optimization for the class B problem size with five particles per cell ($micell = 5$).

and finally multiple sockets on the SMP. We do not present scalability within a core, but rather present the data starting with fully-threaded core. As the vertical axis is performance per thread, perfect scaling would be a horizontal line. However, for the reference MPI implementation, we do not bind processes to cores. The results correspond to default OS scheduling.

On the x86 architectures, we generally observe very good scalability for the Pthreads implementations. The schemes that perform some or full grid replication provide similar performance, and substantially better performance than the non-replicated shared grid. This generally suggest that neither memory bandwidth nor cache capacity are adversely impeding performance for this problem configuration.

The Pthreads fully shared grid implementation demonstrates the best scaling on all the systems, but per-thread performance is affected by synchronization overhead. The grid decompositions reduce updates to the shared grid, but lead to per-thread execution time variability due to the Larmor radius values. The primary problem is that the Larmor radius is independent of the particle's radial location. Thus, threads which are assigned outer annuli in the grid decomposition scheme have more updates into the shared grid and consequently a higher execution time. However the address calculation stage performing 110 flops, scales well on all concurrencies and there is still an overall performance improvement on the Victoria Falls system.

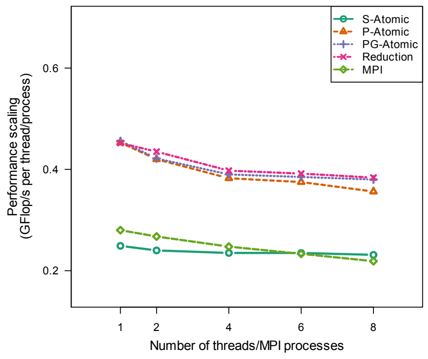
Although Barcelona's MPI scalability shows only a moderate degradation in performance, Nehalem's per core MPI performance drops by more than a factor of three suggesting that the two-way SMT is not of much benefit. Moreover, on both machines, the Pthreads implementations deliver superior performance compared with the MPI implementation.

Victoria Falls demonstrates interesting results. As advertised, chip multithreading (CMT) is designed to exploit all forms of parallelism with a single programming paradigm. Using one core, we see performance on Victoria Falls (8 threads) is comparable to the complex superscalar architectures — an apparent win for CMT. One might expect muticore scalability to be much more readily achieved than multithreading scalability. However, due to the imbalance imposed by the gyrokinetic averaging scheme, we see a sharp drop in performance of the replicated grid approaches beyond 32 threads. Given that this problem instance is fairly small, the reduction step in the replicate and reduce algorithm does not scale beyond 48 threads. Initially the Victoria Falls MPI implementation shows the best performance, likely due to eight processes being scheduled on different cores. However, as the process/thread count increases, the MPI performance drops precipitously, and plummets below the Pthreads implementations. Worse still, the MPI implementation fails to run at concurrencies greater than 64 processes.

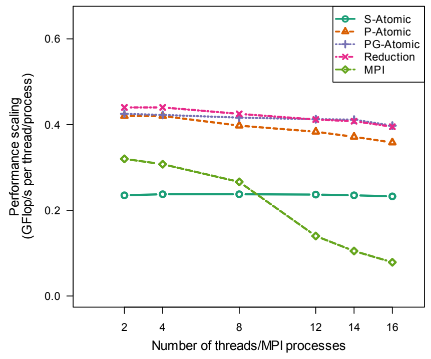
5.3 Memory Utilization

We do not believe that aggregate memory capacity or bandwidth can scale linearly with the number of cores. As such, an algorithm's memory utilization may become a prohibitive factor in software/hardware co-design. To that end, Figure 5 presents the memory utilization for each grid decomposition/synchronization combination for the three architectures. For threaded implementations, auxiliary grid arrays are shared by all threads, whereas in the MPI version, where each task is a separate process, these grid arrays are replicate by the number of processes. When it comes to the charge deposition grid, we observe the four decomposition strategies result in dramatically different memory utilization, but the locking strategies result in tiny changes. A single shared grid requires only a fraction of the memory used by the auxiliary grid arrays. In fact, when using the partitioned grid approach, the aggregate charge deposition grid memory footprint is less than the memory footprint of the auxiliary grid arrays. Total grid replication on thread implementations will be less than double the memory footprint only on SMPs with less than four cores. As we passed that point a few years ago, development of memory-efficient implementations is becoming increasingly important. Replication on threaded implementations use as much memory for the charge deposition grid as replication on MPI implementations, but will use dramatically ($\frac{1}{N_{threads}}$) less memory for the common grids.

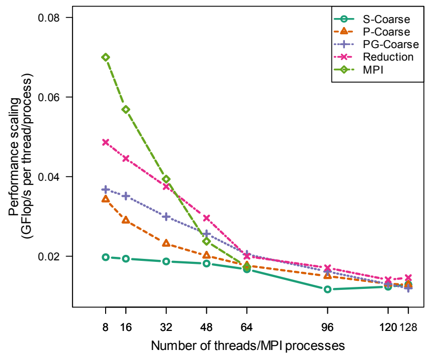
Although 100 MB – 1500 MB might not sound like much on the Victoria Falls system, one should be mindful that Figure 5 only shows the memory capacity for the small, low density problem. As the problem size and density increase, so too will the memory footprint. When extrapolated into the future, exponentially increasing core counts cast serious doubts on the viability of grid replication implementations.



(a) Barcelona parallel scaling

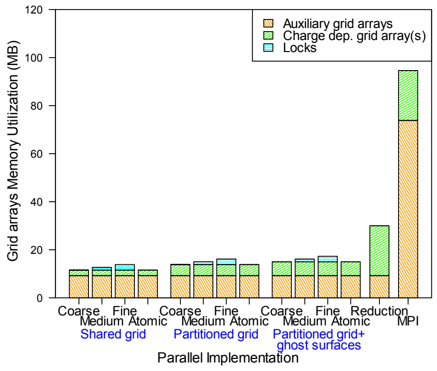


(b) Nehalem parallel scaling

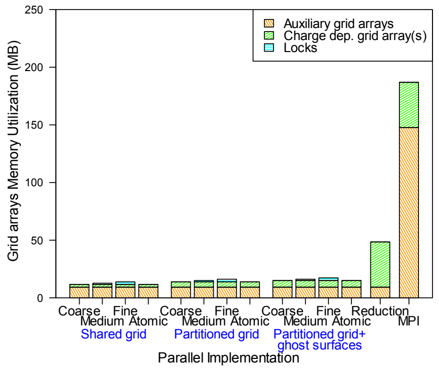


(c) Victoria Falls parallel scaling

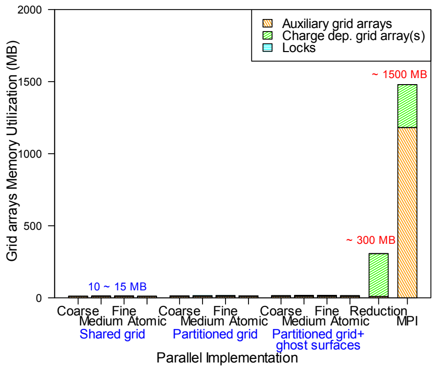
Figure 4: Best performance as a function of grid decomposition for the class B problem size with five particles per cell ($micell = 5$) using the best synchronization method.



(a) Barcelona grid memory footprint



(b) Nehalem grid memory footprint



(c) Victoria Falls grid memory footprint

Figure 5: Memory footprint as a function of grid decomposition, and synchronization for the class B problem size with five particles per cell ($micell = 5$).

5.4 Performance of Memory-Efficient Implementations

Figure 6 shows the GFLOP/s rate as a function of particle density and problem size using the best, memory-efficient threaded implementation for that particular configuration. We do not consider replicate and reduce to be memory-efficient, as the memory footprint scales linearly with the number of threads. Nevertheless, the largest problems and highest densities coupled with the most memory-efficient threaded implementations still require more memory than is installed on our SMPs.

Larger problem sizes demand larger caches, but larger particle densities can yield higher temporal locality. As such, we observe that performance decreases with increasing grid size, and increases with increasing particle density, albeit often by only a factor of two. Victoria Falls balks at this trend for small problem sizes (A in this case) and low particle densities. This likely occurs due to the fact that $N_{threads}$ exceeds *mpsi*. As a result, simple partitioned grid strategies leave some threads without grid points. Fine partitioning ameliorates this, but the locking strategies limit potential concurrency.

Figure 7 details the relative performance of our best memory-efficient threaded implementation to the best replicate and reduce threaded implementation. When the number is close to 1.0×, we attain near-optimal performance without squandering memory. Note that it is possible for the performance of memory-efficient threaded implementations to exceed the reduction approach. In such cases, we deliver both optimal performance and minimize the memory footprint.

On Barcelona, the memory-efficient approach yields better than 80% of the best performance using roughly 45% of the memory footprint. The results on Nehalem are even more encouraging — typically achieving better than 95% of the best performance while using only 33% of the memory footprint of the threaded replicate and reduce implementation. We observe that for *micell* values, the memory-efficient approach performs better than the reduction, as the overhead of reduction in comparison to the density updates is proportionally higher.

As particle density increases on Victoria Falls, the memory-efficient performance falls away from the reduction approach regardless of problem size. In effect, the reduction cost has been amortized by the sheer number of charge depositions.

5.5 Performance Impact of Random Particle Distributions

Thus far, all quoted performance numbers have relied on a (partially) sorted particle distribution. However, without continual or periodic sorting, the particle positions will slowly be randomized. As such, we must examine the performance on a random distribution to motivate periodic partial sorting. To that end, Figure 8 presents the fraction of performance (Figure 6) of the best memory-efficient implementation as a function of problem configuration when the particle distribution is randomized. Generally, randomized particle locations force threads to update the shared grid far more frequently, as there is no longer a correlation between the thread performing the update and the grid points to be updated.

Clearly, the drop in performance is highly correlated with problem size, and only moderately so with particle density. On Barcelona, we observe that random particle distributions on the mid-sized problems show the most profound drop in performance likely due to the per-core increased cache capacity requirements. As the cache capacity requirements for the large problem likely already exceed the core's caches, they show little additional performance drops. Similarly, the smallest problems easily fit in the last level cache and thus don't show as profound of a performance drop. A similar, but more significant trend exists on Nehalem.

Figure 9 shows the relative performance of the memory-efficient approach to the reduction approach for a randomized particle distribution. Unlike Figure 7, where the two were nearly equal, we observe that the reduction approach consistently, and dramatically outperforms the memory-efficient approach on all architectures and problem configurations — up to 2× on x86, and 5× on Victoria Falls. As such, we believe in the future one must balance the benefits of reductions with the costs of sorting and additional memory capacity.

5.6 Performance Advantage of Threaded Implementations

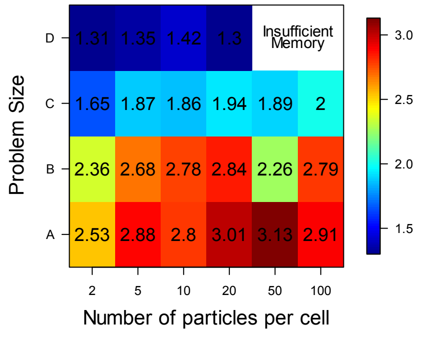
Finally, we examine the performance advantage of our threaded implementation over the existing MPI implementation across problem sizes using a sorted particle distribution.

If memory utilization is not a concern, then we may include the threaded replicate and reduce implementation in our comparison. As shown in Figure 10, our threaded implementation exceeds MPI performance for all architectures and all problem configurations. Generally the advantage is diminished as particle density increases — i.e. MPI reduction time is effectively amortized. Atomic operations provide a substantial advantage over MPI reductions for low particle densities — 1.9×, 4.4×, and 2.3× better performance on Barcelona, Nehalem, and Victoria Falls respectively.

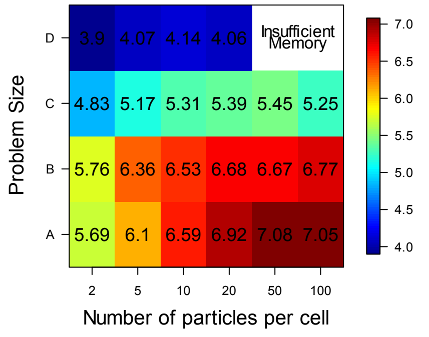
In the future, we do not believe replication strategies will be cost-effective. As such, Figure 11 compares our best memory-efficient threaded implementation with the MPI reduction approach. Although we have algorithmically disadvantaged the threaded implementation, it not only consistently delivers superior performance on Barcelona and Nehalem, but uses dramatically less memory. However, as density increases beyond 5 particles per grid point, Victoria Falls' memory-efficient threaded performance dips below MPI eventually reaching roughly 65% MPI's performance. We believe that in the future, designers must balance performance (algorithms) with memory capacity (design and cost) for high concurrency multicore processors.

6. CONCLUSIONS

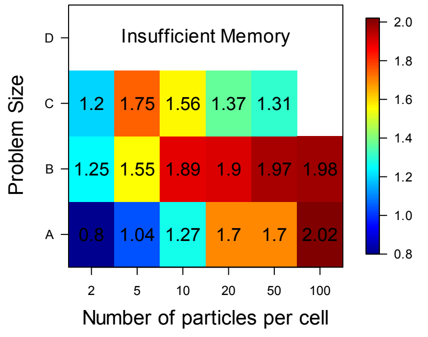
In this paper, we examined the benefits of a memory-efficient, threaded multicore implementation of the charge deposition kernel of the GTC application when compared with the traditional replicate and reduce MPI implementation. Without regard to memory utilization, we typically see performance gains in excess of 1.5× and as high as 4.4×. However, we note that one of the limiting factors of the MPI implementation is its high memory requirements. To that end, we examined several memory-efficient implementations that both dramatically reduce the memory usage, and ensure the memory usage remains constant as thread-level parallelism scales. On x86 architectures, such implementations deliver near optimal performance at a fraction of the memory usage. We realize such gains through the elimination



(a) Barcelona

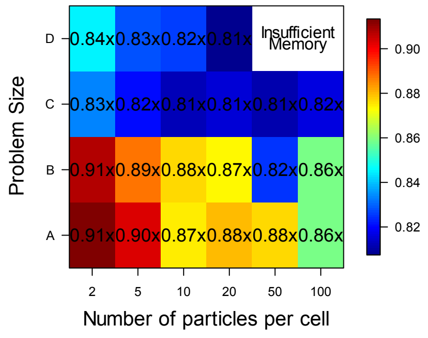


(b) Nehalem

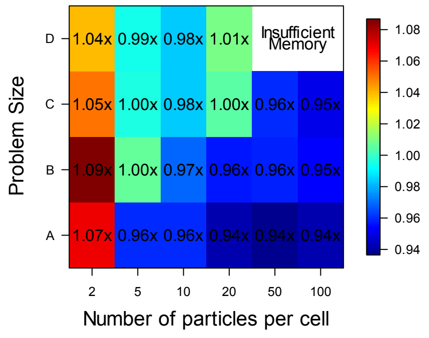


(c) Victoria Falls

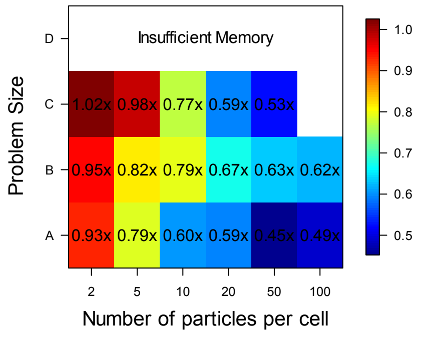
Figure 6: Parallel performance (in GFlop/s) achieved by the best memory-efficient charge deposition variant for various grid size and particles-per-cell configurations.



(a) Barcelona

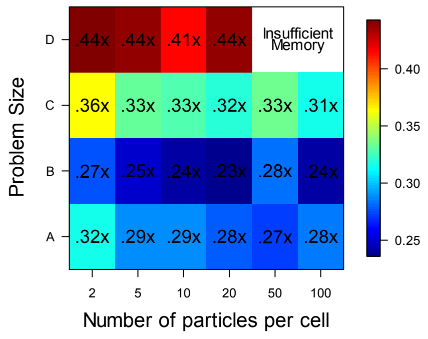


(b) Nehalem

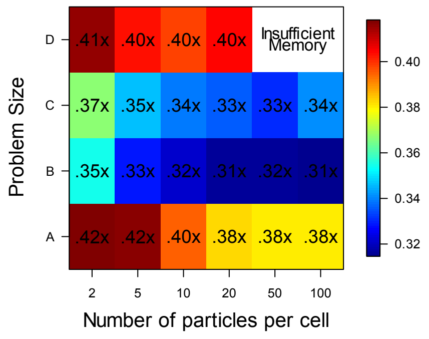


(c) Victoria Falls

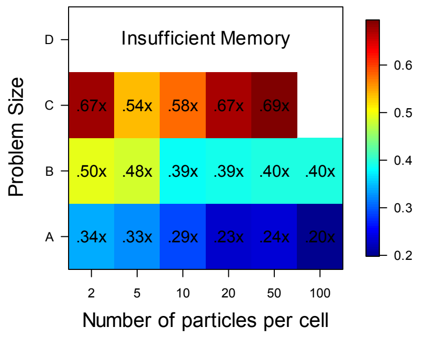
Figure 7: The ratio of the performance of the best memory-efficient charge deposition variant to the Reduction approach.



(a) Barcelona

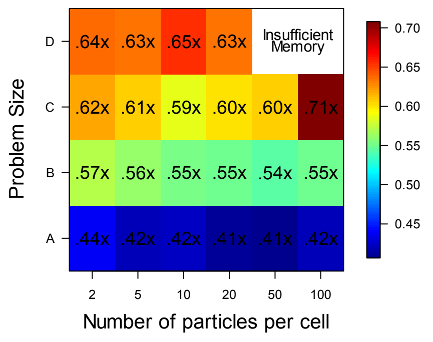


(b) Nehalem

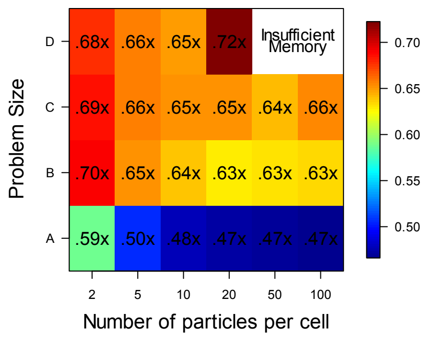


(c) Victoria Falls

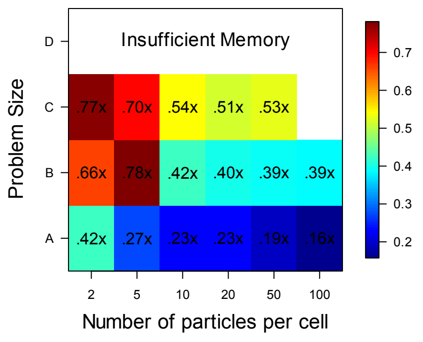
Figure 8: The impact of a randomized particle distribution (multiplicative factor, compared to Figure 6) on the performance of the best memory-efficient charge deposition variant.



(a) Barcelona

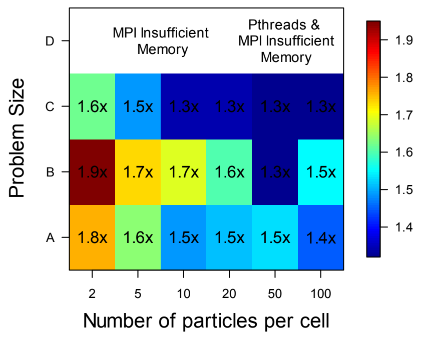


(b) Nehalem

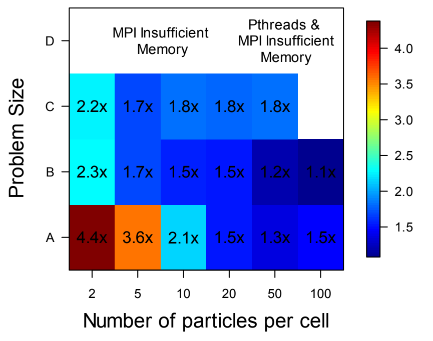


(c) Victoria Falls

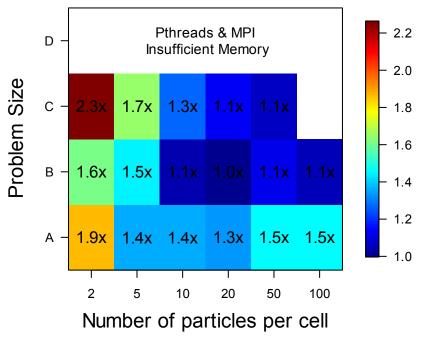
Figure 9: The ratio of the performance of the best memory-efficient charge deposition variant to the Reduction approach for a randomized particle distribution.



(a) Barcelona

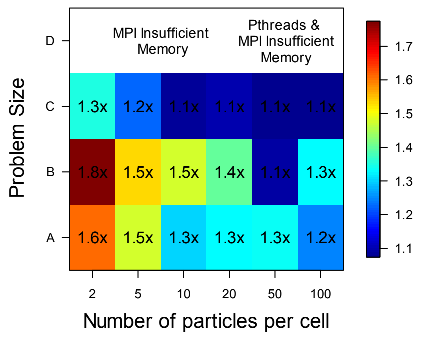


(b) Nehalem

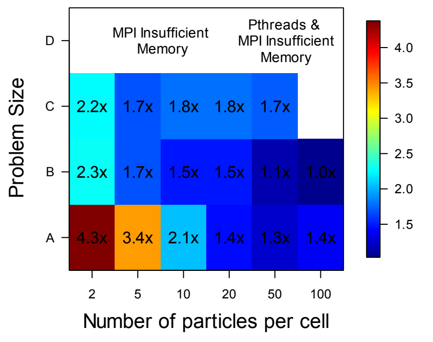


(c) Victoria Falls

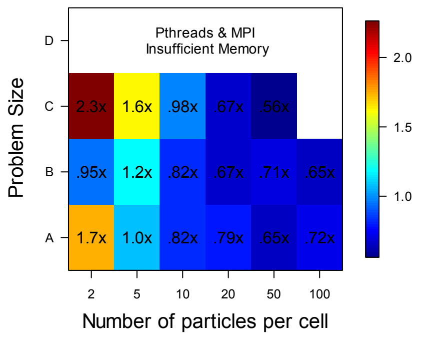
Figure 10: Speedup achieved by the best Pthreads implementation over the reference MPI code for various grid size and particles-per-cell configurations.



(a) Barcelona



(b) Nehalem



(c) Victoria Falls

Figure 11: Speedup achieved by the best memory-efficient Pthreads implementation over the reference MPI code for various grid size and particles-per-cell configurations.

of massive N-way reductions, data structure reorganization, faster synchronization, and possibly the elimination of MPI overhead.

Perhaps the biggest concern is that the maximum Larmor radius value scales roughly as $\frac{mpsi}{16}$ for the GTC problem sizes we studied. As such, the straightforward partitioned grid approach will not scale linearly beyond 16 threads, as charge rings more frequently straddle multiple annuli. Under such conditions, threads cannot update their exclusive grids, but rather must update the shared grid — a slower operation. As concurrency increases, the updates to the shared grid predominate.

To solve this problem, future work will explore both the middle ground between the full N-way grid replication of the MPI implementation and the 2-way replication of our partitioned grid approach. We will also examine 2D decompositions, particle binning to improve cache performance, and GTC inter-kernel optimizations.

We observe that since abandoning the front-side bus architecture, Intel's new Nehalem processor doubles the performance of AMD's Barcelona and triples the performance of Sun's Niagara2. However, the processor is far more sensitive to a lack of process pinning. On all three systems, faster atomic increments would boost the performance of all the Pthreads implementations, and also ameliorate the scalability issues with the partitioned grid schemes due to the Larmor radius variation.

Acknowledgments

We would like to express our gratitude to Intel and Sun for their hardware donations. This work was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231, Microsoft (Award #024263), Intel (Award #024894), and by matching funding through U.C. Discovery (Award #DIG07-10227).

7. REFERENCES

[1] M.F. Adams, S. Ethier, and N. Wichmann. Performance of particle in cell methods on highly concurrent computational architectures. *Journal of Physics: Conference Series*, 78:012001 (10pp), 2007.

[2] K.J. Bowers. Accelerating a particle-in-cell simulation using a hybrid counting sort. *Journal of Computational Physics*, 173(2):393–411, 2001.

[3] K.J. Bowers, B.J. Albright, B. Bergen, L. Yin, K.J. Barker, and D.J. Kerbyson. 0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner. In *Proc. 2008 ACM/IEEE Conf. on Supercomputing*, pages 1–11, Austin, TX, November 2008. IEEE Press.

[4] S. Ethier, W.M. Tang, and Z. Lin. Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. *Journal of Physics: Conference Series*, 16:1–15, 2005.

[5] S. Ethier, W.M. Tang, R. Walkup, and L. Oliker. Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas. *IBM Journal of Research and Development*, 52(1-2):105–116, 2008.

[6] W.W. Lee. Gyrokinetic particle simulation model. *Journal of Computational Physics*, 72(1):243–269, 1987.

[7] Z. Lin, T.S. Hahm, W.W. Lee, W.M. Tang, and R.B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 281(5384):1835–1837, 1998.

[8] G. Marin, G. Jin, and J. Mellor-Crummey. Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code. *Journal of Physics: Conference Series*, 125:012087 (6pp), 2008.

[9] L. Oliker, A. Canning, J. Carter, C. Iancu, M. Lijewski, S. Kamil, J. Shalf, H. Shan, E. Strohmaier, S. Ethier, and T. Goodale. Scientific application performance on candidate petascale platforms. In *Proc. 21st Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–12, Long Beach, CA, March 2007. IEEE.

[10] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Proc. 2004 ACM/IEEE Conf. on Supercomputing*, page 10, Pittsburgh, PA, November 2004. IEEE Computer Society.

[11] G. Stantchev, W. Dorland, and N. Gumerov. Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. *Journal of Parallel and Distributed Computing*, 68(10):1339–1349, 2008.